On Robust Malware Classifiers by Verifying Unwanted Behaviours

On Robust Malware Classifiers by Verifying Unwanted Behaviours

Wei Chen[1], David Aspinall[1], Andrew D. Gordon[1,2], Charles Sutton[1], and Igor Muttik[3]

[1] University of Edinburgh, UK,
{wchen2,csutton}@inf.ed.ac.uk,
{David.Aspinall,Andy.Gordon}@ed.ac.uk
[2] Microsoft Research Cambridge, UK,
[3] Intel Security, UK,
igor.muttik@intel.com

Abstract. Machine-learning-based Android malware classifiers perform badly on the detection of new malware, in particular, when they take API calls and permissions as input features, which are the best performing features known so far. This is mainly because signature-based features are very sensitive to the training data and cannot capture general behaviours of identified malware. To improve the robustness of classifiers, we study the problem of learning and verifying *unwanted behaviours* abstracted as automata. They are common patterns shared by malware instances but rarely seen in benign applications, e.g., intercepting and forwarding incoming SMS messages. We show that by taking the verification results against unwanted behaviours as input features, the classification performance of detecting new malware is improved dramatically. In particular, the precision and recall are respectively 8 and 51 points better than those using API calls and permissions, measured against industrial datasets collected across several years. Our approach integrates several methods: formal methods, machine learning and text mining techniques. It is the first to automatically generate unwanted behaviours for Android malware detection. We also demonstrate unwanted behaviours constructed for well-known malware families. They compare well to those described in human-authored descriptions of these families.

Keywords: mobile security, static analysis, software verification, machine learning, malware detection

1 Introduction

Android malware, including trojans, spyware and other kinds of unwanted software, has been increasingly seen in the wild and even on official app stores [17, 37]. To automatically detect Android malware, machine learning methods have been applied to train malware classifiers [5, 8, 21, 22, 36]. Among them, the tool Drebin [8] extracts a broad range of features, such as permissions, components, API calls and intents, then trains an SVM classifier. DroidAPIMiner [5] uses

refined API calls and relies on the KNN (k-nearest neighbours) algorithm. Another interesting tool is CHABADA [22] which detects outliers (abnormal API usage) within clusters of applications by exploiting OC-SVM (one-class SVM). All of these classifiers were trying to obtain good fits to the training data by using different methods and variant kinds of features. However, the robustness of malware classifiers has received much less consideration. As we will show in Table 2, the classification performance of detecting new malware is poor, in particular, when API calls and permissions are used as input features, which are the most popular and the best performing features known so far.

On the other hand, researchers and malware analysts have organised malware instances into hundreds of families [30, 37], e.g., Basebridge, Geinimi, Ginmaster, Spitmo, Zitmo, etc. These malware instances share certain unwanted behaviours, for example, sending premium messages constantly, collecting personal information, loading classes from hidden payloads then executing commands from remote servers, and so on. Except for some inaccurate online analysis reports [1–4, 24] of identified malware families, however, people have no idea of what exactly happens in these malware instances.

We want to learn unwanted behaviours exhibited in hundreds and thousands of malware instances and verify the application in question, e.g., an application submitted to an app store, to deny them. We will show that verifying these unwanted behaviours can improve the robustness of Android malware classifiers. Our approach integrates formal methods, machine learning, and text mining techniques, and proceeds as follows.

- **Formalisation**. We approximate an Android application's behaviours by a finite-state automaton, that is, a set of finite control-sequences of events, actions, and annotated API calls. Since different API calls might indicate the same behaviour, we abstract the automaton by aggregating API calls into permission-like phrases. We call it a *behaviour automaton*.
- **Learning**. An *unwanted behaviour* is a common behaviour which is shared by malware instances and has been rarely seen in benign applications. We develop a machine-learning-centred method to infer unwanted behaviours, by efficiently constructing and selecting sub-automata from behaviour automata of malware instances. This process is guided by the behavioural difference between malware and benign applications.
- **Refinement**. To purify unwanted behaviours, we exploit the family names of malware instances to help figure out the most informative unwanted behaviours. We compare unwanted behaviours with the human-authored descriptions for malware families, to confirm that they match well with patterns described in these descriptions.
- **Verification**. We check whether the application in question has any security fault by verifying whether the intersection between its behaviour automaton and an unwanted behaviour is not empty.

We take malware instances released in different years respectively as training, validation and testing sets. They were collected from several industrial datasets.

- **Training & Validation.** We collected 3,000 malware instances, which have been discovered between 2011 and 2013, and 3,000 benign applications. They include some famous benign applications, such as Google Talk, Amazon Kindle, and Youtube, and so on; and all malware instances from Malware Genome Project [37] and most malware instances from Mobile-Sandbox [30]. These malware instances have been manually investigated and organised into around 200 families by third-party researchers and malware analysts. By reading their online malware analysis reports [1–4, 24], we learned what bad things would happen in these malware instances. We divided them into a training set and a validation set. Each of them consists of 1,500 malware instances across all families and 1,500 benign applications.
- **Testing.** We test using a collection of 1,500 malware instances, which were released in 2014, and 1,500 benign applications. These malware instances were from Intel Security and have been investigated by malware analysts. But, there is no family information or online analysis report for them. We have no idea of their unwanted behaviours. The collection of benign applications, which was collected in 2014, is disjoint from the collection of benign applications used for training and validation, which was collected between 2011 and 2013. These two collections were all supplied by Intel Security.

We use API calls, permissions, and the verification results against unwanted behaviours as input features; then apply L1-Regularized Linear Regression [32] to train classifiers. The evaluation on the testing set shows that the precision and recall of using unwanted behaviours are respectively 8 and 51 points better than those of using API calls and permissions. As shown in Table 2, using API calls and permissions as input features, can achieve very good precision and recall on the validation set, however, its classification performance on the testing set is poor. That is, unwanted behaviours are more general than API calls and permissions. This is needed in practice: to mitigate over-fitting and improve the robustness of malware classifiers.

Our approach is the first to learn unwanted behaviours from Android malware instances. The main contributions of this paper are to:

- demonstrate that it is hard to detect new malware for classifiers trained on identified Android malware instances, by using signature-based features;
- show that using semantics-based features like unwanted behaviours dramatically improves the classification performance of new malware detection;
- supply a static analysis tool to construct behaviour automata from the bytecode, considering a broad range of features of the Android framework;
- apply a novel machine-learning-centred algorithm to efficiently choose salient sub-automata to characterise unwanted behaviours;
- apply a refinement approach to look up the most informative unwanted behaviours, by making use of the family names of malware instances.

Related Work. The idea to abstract applications' behaviours as automata is similar with the behaviour abstraction in [11, 34]. The behaviour automata are close to permission-event graphs [16], and more compact than embedded call

graphs [21] and behaviour graphs [35]. None of them has been exploited to automatically generate verifiable properties.

The idea to learn unwanted behaviours is close to the methodology proposed by Fredrikson et al. to synthesize malware specification [19]. In their work, a data dependence graph with logic constraints on nodes and edges was used to characterise an application's behaviours. From the graphs of malware instances and benign applications they constructed so-called significant subgraphs that maximise the information gain. Then, the optimal collections of subgraphs were selected using the formal concept analysis. The main drawback of this method is its scalability. Also, the training and testing sets were very unbalanced, i.e., the number of benign applications is much less than that of malware instances. We overcome these limitations by using behaviour automata as the abstract model, training and testing on large and balanced datasets.

The unwanted behaviours can be considered as instances of security automata [28]. Our verification approach is the same as the automata-theoretic model checking [33]. In total, 19 malicious properties for Android applications were manually constructed and specified as first-order LTL formulae in [23]. Some benign and malicious properties specified in LTL were verified against hundreds of Android applications in [16]. But, none of these properties was automatically constructed.

Among others, Angluin's [7] and Biermann's [12] algorithms were developed to learn regular expressions from sample finite strings. To apply similar ideas in unwanted behaviour construction, we have to extract enough finite strings from applications to approximate their behaviours. Compared with our construction of behaviour automata, this would be more complex and expensive.

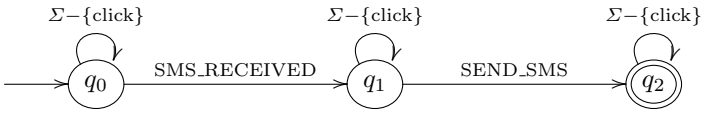
2 An Example Unwanted Behaviour

Let us consider a malware family called Ggtracker. A brief human-authored description of this family which was produced by Symantec [4] is as follows.

> *It sends SMS messages to a premium-rate number. It monitors received SMS messages and intercepts SMS messages. It may also steal information from the device.*

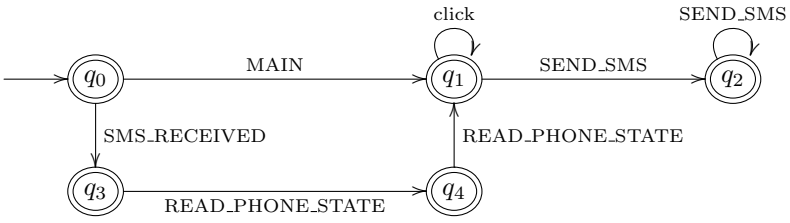
One of the unwanted behaviours we have learned from malware instances in this family can be expressed as the regular expression: SMS_RECEIVED.SEND_SMS. The approach to learn these unwanted behaviours will be elaborated in Section 4. It denotes the behaviour of sending an SMS message out *immediately* after an incoming SMS message is received without the interaction from the user. Some behaviours of the application in question are not the same as the unwanted behaviours, but, they often have the unwanted behaviours as sub-sequences. For example, the behaviour SMS_RECEIVED.READ_PHONE_STATE.SEND_SMS contains SMS_RECEIVED.SEND_SMS as a subsequence. To capture behaviours sharing the same patterns with the unwanted behaviours, if a behaviour contains an unwanted behaviour as a sub-sequence, we consider this behaviour as

unwanted as well. We call them *extended* unwanted behaviours. So, we generalise from the above unwanted behaviour and construct the following automaton ψ:



Here, we use the symbol Σ to denote the collection of events, actions, and permission-like phrases and the word "click" to denote that there is no interaction from the user. In Section 4.2 we will show a method to refine unwanted behaviours by making use of the family names of malware instances. To distinguish and compare these unwanted behaviours, we use respectively *unwanted*, *ext. unwanted*, and *ext. unwanted for families* to denote them.

We now want to verify whether a target application has the above unwanted behaviour. Let us consider the following behaviour automaton $\mathcal{A}$:



It is constructed from the bytecode of an Android application using static analysis. Its source code and the method to construct behaviour automata will be given in Section 3. It tells us: this application has two entries which are respectively specified by actions MAIN and SMS_RECEIVED; it will collect information like the phone state, then send SMS messages out; the behaviour of sending SMS messages can also be triggered by an interaction from the user, e.g., click a button, touch the screen, long-press a picture, etc., which is denoted by the word "click". A string accepted by this automaton characterises a behaviour of this application. All states in this automaton are accepting states since any prefix of an application's behaviours is one of its behaviours as well.

Because the intersection between $\mathcal{A}$ and ψ is not empty, we consider this application is unsafe with respect to the unwanted behaviour ψ. In Section 5, we will show that this verification against unwanted behaviours can improve the classification performance of new malware detecting.

3 Behaviour Automata

We use a simplified synthetic application to illustrate the construction of behaviour automata.

3.1 An Example Android Application

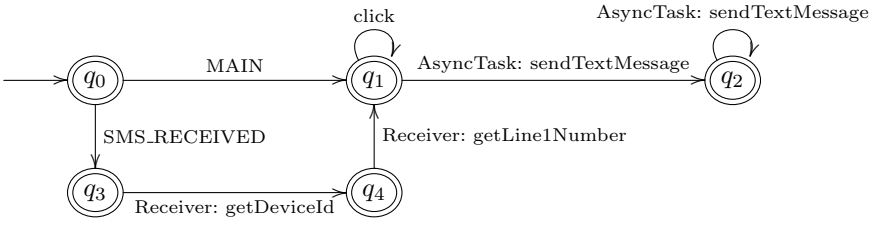
This application will constantly send out the device ID and the phone number by SMS messages in the background when an incoming SMS message is received. Its source code and part of its manifest file follow.

```
public class Main extends                                    /* Main.java */
  Activity implements View.OnClickListener {
  private static String info = "";
  protected void onCreate(Bundle savedInstanceState) {
    Intent intent = getIntent();
    info = intent.getStringExtra("DEVICE_ID");
    info += intent.getStringExtra("TEL_NUM");
    SendSMSTask task = new SendSMSTask();
    task.execute(); }
  public void onClick (View v) {
    SendSMSTask task = new SendSMSTask();
    task.execute(); }
  private class SendSMSTask extends AsyncTask<Void, Void, Void> {
    protected Void doInBackground(Void... params) {
      while (true) {
        SmsManager sms = SmsManager.getDefault();
        sms.sendTextMessage("1234", null, info, null, null); }
      return null; }}}

public class Receiver extends BroadcastReceiver {          /* Receiver.java */
  public void onReceive(Context context, Intent intent) {
    Intent intent = new Intent();
    intent.setAction("com.main.intent");
    TelephonyManager tm = (TelephonyManager)
    getBaseContext().getSystemService(Context.TELEPHONY_SERVICE);
    intent.putExtra("DEVICE_ID", tm.getDeviceId());
    intent.putExtra("TEL_NUM", tm.getLine1Number());
    sendBroadcast(intent); }}

                                                /* AndroidManifest.xml */
<activity android:name="com.example.Main" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <action android:name="com.main.intent" />
  </intent-filter>
</activity>
<receiver android:name="com.example.Receiver" >
  <intent-filter>
    <action android:name="android.provider.Telephony.SMS_RECEIVED" />
  </intent-filter>
</receiver>
```

As specified in AndroidManifest.xml, the Main activity can handle a specific Intent called "com.main.intent" and the Receiver will be triggered by an incoming SMS message (SMS_RECEIVED). After the Receiver collects the device ID and the phone number, it will send them out by a broadcast with the intent "com.main.intent". This broadcast is then handled by the Main activity in the method onCreate. Afterwards, SMS messages containing the device ID and the phone number are sent out in the background in an AsyncTask.

3.2 An Example Behaviour Automaton

From the bytecode of this application, we construct the following automaton.



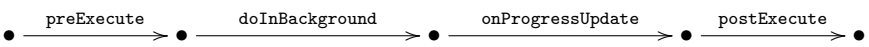
This automaton is constructed from finite traces of actions, events, and annotated API calls using static analysis. Actions reflect what happens in the environment and what kind of service an application requests for, e.g., an incoming message is received, the device finishes booting, the application wants to send an email by using the service supplied by an email-client, etc. Events denote the interaction from the user, e.g., clicking a picture, pressing a button, scrolling down the screen, etc. Annotated API calls tell us whether the application does anything we are interested in. For instance, getDeviceID, getLine1Number, and sendTextMessage are annotated API calls in the above example.

For a single behaviour there are often several related API methods. For example, getDeviceId, getLine1Number, and getSimSerialNumber are all related to the behaviour of reading phone state. We categorise API methods into a set of permission-like phrases, which describe behaviours of applications, so as to remove redundancy caused by API calls which indicate the same behaviour. This results in an abstract automaton, so-called a *behaviour automaton*. It has several advantages, including: more resilient to variants of behaviours, such as swapping two API calls related to the same behaviour; more compact automata, which are good for human-understanding and further analysis, by reducing the number of labels on the edges. For instance, the behaviour automaton for the above example is the automaton $\mathcal{A}$ depicted in Section 2.

3.3 The Implementaion

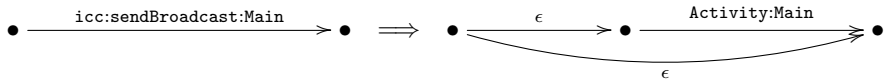
In our implementation, we use an extension of permission-governed API methods generated by PScout [9] as annotations. The Android platform tools `aapt` and `dexdump` are respectively used to extract the manifest information and to decompile the bytecode into the assembly code, from which we construct the automaton. It took around two weeks to generate automata for $10,000$ applications using a multi-core desktop computer. More technical details are as follows.

- *Multiple Entries.* A class becomes an entry if a system action, e.g., MAIN and SMS_RECEIVED, has been declared as one of its intent-filters in the manifest file. For developer-defined actions, e.g., "com.main.intent" in the earlier example, their corresponding classes become entries only when an instance of a class is explicitly created at some control-reachable point from a system entry.
- *Class Exploration.* Starting from an entry class, from the callbacks of each visited class, we collect new classes by exploring the new-instance and start-component relations.
- *Component life-cycle.* We organise the callbacks in each Android component according to its life-cycle, e.g., the life-cycle of AsyncTask is modelled as:



- *Inter-Procedural Calls.* We build an inter-procedural call graph for each callback in each reachable class.

– *New-Instances and Start-Components.* For each new-instance and each start-component relation, we add an ϵ-transition from the entry-point of new-instance or start-component statement to the entry-point of the call graph for the target class. The original transitions for these statements in the caller's call graphs are replaced by ϵ-transitions. Intuitively, this models the asynchronisation by non-deterministic choices. For example, for the statement sendBroadcast in onReceive method, we have:



– *Callbacks.* We collect the following functions as callbacks: overridden methods of Android component classes, implementations of abstract functions declared in listener interfaces, and callbacks defined in layout files.
– *Inter-Component Communications.* We search through all methods for start-component API calls, e.g., startService, startActivity, sendBroadcast, etc. We decide whether there is a data-flow from a register containing a constant to the entry-point of a start-component statement, and if so, we decide whether this constant is a system action, a developer-defined action, or a developer-defined class name. For the first two, we search the manifest file for possible matched components. The last one has been dealt as a start-component relation in the class exploration.

We don't model registers, fields, assignments, operators, pointer-aliases, arrays or exceptions. The choice of which features to model is a trade-off between efficiency and precision.

4 Learning and Refining Unwanted Behaviours

Once a behaviour automaton has been constructed for each malware instance, we want to capture the common behaviour shared by malware, which is rarely seen in benign applications, so-called an *unwanted behaviour*.

4.1 Salient Sub-Automata

The space of candidate behaviours, which consists of the intersection and difference between behaviour automata, in theory, is exponential in the number of sample applications. To combat this, we approximate this space by searching for a "salient" subspace. The searching process is guided by the behavioural difference between malware and benign applications. We formalise this process as the algorithm in Fig. 1.

The main process construct_features takes a collection G of behaviour automata as input and outputs a set F of salient sub-automata with their weights W. Here, a sub-automaton is *salient* if it is actually used in a linear classifier, i.e., its weight is not zero.

We divide G into N groups: $G_0, \ldots, G_i, \ldots, G_{N-1}$. For each group, we construct sub-automata by computing the intersection and difference between automata within this group, i.e., merge_features $(G_i, \emptyset)$. This results in N feature sets $F_0, \ldots, F_i, \ldots, F_{N-1}$. The sub-automata in each set are disjoint. Then, we merge sub-automata from different groups, i.e., merge_features (G_i, G_j). This process stops when all groups have been merged into a single group.

```
Function: construct_features (G, α)
Input: G – a group of behaviour automata
       α – the lower bound on the classification accuracy
Output: salient sub-automata and their weights
 1:  G_{i∈[0..N−1]} ← divide the set G into N groups
 2:  for i ∈ [0..N − 1]
 3:    F_i ← merge_features (G_i, ∅)
 4:  for (s ← 2; s ≤ N; s ← s × 2)
 5:    for i ∈ [s − 1..N − 1]
 6:      j ← i − (s/2)
 7:      (F_i, _), (F_j, _) ← diff_features (F_i, α), diff_features (F_j, α)
 8:      if (i + 1)%s = 0 then
 9:        F_i ← merge_features (F_i, F_j)
10:      elif (i + 1) > (N/s) × s and (i + 1)%(s/2) = 0 then
11:        F_j ← merge_features (F_i, F_j)
12:  return diff_features (F_{s/2−1}, α)
Function: merge_features (E, F)
 1:  for e ∈ E
 2:    for f ∈ F
 3:      if f − e ≠ ∅ then F ← F ∪ {f − e}
 4:      if f ∩ e ≠ ∅ then F ← F ∪ {f ∩ e}
 5:      if f − e ≠ f and f ∩ e ≠ f then F ← F − {f}
 6:      e ← e − f
 7:    if e ≠ ∅ then F ← F ∪ {e}
 8:  return F
Function: diff_features(F, α)
 1:  D ← add an equal number of randomly-chosen benign applications
 2:        into the set of malware instances from which F was collected
 3:  W, acc ← train (D, F)
 4:  if acc > α then F ← {f ∈ F | W_f ≠ 0}
 5:  return F, W
```

Fig. 1. The algorithm for the construction of salient sub-automata.

Before merging sub-automata from two different groups, for each group, we train a linear classifier, i.e., train (D, F), using a training set D and a feature set F. This training set consists of behaviour automata of malware instances in the group and an equal number of behaviour automata of randomly-chosen benign applications. The input feature set F consists of disjoint sub-automata, which are constructed from behaviour automata of malware instances in the group. Then,

if the classification accuracy acc on the training set is above a lower bound α, we return sub-automata with non-zero weights. Otherwise, we return all features in F. This process differentiates salient features by adding benign applications. It is formalised as the function diff_features.

In our implementation, we adopt L1-Regularized Logistic Regression [32] as the training method. This is because this method is specially designed to use fewer features. The lower bound α on the classification accuracy is set to 90%. We put malware instances from the same malware family into one group so that the searching process is more directed. We have also designed and implemented a multi-process program to accelerate the construction, i.e., construct sub-automata for each group simultaneously. It took around one week to process $4,000$ malware instances using a multi-core desktop computer. At the end of the computation, we produced around $1,000$ salient sub-automata.

4.2 Refinement

We will use these salient sub-automata to characterise unwanted behaviours. A straightforward way is to choose automata by their weights, for example, those with negative weights, i.e., $\{f \in F \mid W_f < 0\}$. To purify unwanted behaviours, we want to exploit the family names of malware instances to figure out the most informative ones, that is, to choose a small set of salient sub-automata to characterise unwanted behaviours for each family. Here are several candidate methods.

- *Top-n-negative*. For a linear classifier, intuitively, a feature with a negative weight more likely indicates an unwanted behaviour, and a feature with a positive weight more likely indicates a normal behaviour. This observation leads us to refine unwanted behaviours by using sub-automata with negative weights, i.e., choose the top-n features from the set $\{f \in F \mid W_f < 0\}$ by ranking the absolute values of their weights.
- *Subset-search*. For each malware family, we choose a subset X of salient sub-automata, such that it largely covers and is strongly associated with malware instances in this family. Formally, we use $Pr(f|X)$ to denote the probability that a malware instance belonging to a family f if all automata in X are sub-automata of the behaviour automaton of this instance, and $Pr(X|f)$ to denote the probability that all automata in X are sub-automata of the behaviour automaton of a malware instance if this instance belongs to f. We use their F_1-measure as the evaluation function to look up subsets. i.e., $\frac{2Pr(f|X)Pr(X|f)}{Pr(f|X)+Pr(X|f)}$. Since exhaustively searching a power-set space is expensive, we adopt Beam Search [25, Chapter 6] to approximate the best K-subsets.
- *TF-IDF*. Another method is to consider features as terms, features from malware instances in a family as a document, and the multi-set of features as the corpus. We rank features by their TF-IDF (term frequency and inverse document frequency) and choose a maximum of m features to characterise unwanted behaviours of each family.

We use the salient sub-automata produced in previous subsection and construct unwanted behaviours for each family by combining all methods discussed earlier. We list human-authored descriptions and learned unwanted behaviours of 10 prevalent families in Table 1. These descriptions for families were collected from their online analysis reports [1–4, 24].

human-authored description	*learned unwanted behaviours in regular expressions*
Arspam. *Sends spam SMS messages to contacts on the compromised device* [4].	1. BOOT_COMPLETED . SEND_SMS
Anserverbot. *Downloads, installs, and executes payloads* [1].	1. UMS_CONNECTED . LOAD_CLASS* . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET) . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET | LOAD_CLASS)*
Basebridge. *Forwards confidential details (SMS, IMSI, IMEI) to a remote server* [2]. *Downloads and installs payloads* [1, 4].	1. UMS_CONNECTED . (INTERNET | LOAD_CLASS | READ_PHONE_STATE | ACCESS_NETWORK_STATE)$^+$
Cosha. *Monitors and sends certain information to a remote location* [4].	1. MAIN . click . (click | ACCESS_FINE_LOCATION | DIAL)* . DIAL . (click | ACCESS_FINE_LOCATION | DIAL)* . (INTERNET | ϵ) 2. SMS_RECEIVED . (INTERNET | ACCESS_FINE_LOCATION)$^+$
Droiddream. *Gains root access, gathers information (device ID, IMEI, IMSI) from an infected mobile phone and connects to several URLs in order to upload this data* [1, 2].	1. PHONE_STATE . (ACCESS_NETWORK_STATE | READ_PHONE_STATE$^+$. INTERNET) . (ACCESS_NETWORK_STATE | INTERNET)*
Geinimi. *Monitors and sends certain information to a remote location* [4]. *Introduces botnet capabilities with clear indications that command and control (C&C) functionality could be a part of the Geinimi code base* [3].	1. ϵ | MAIN . click$^+$. VIBRATE . (click | VIBRATE)* . RESTART_PACKAGES . (MAIN . (click | VIBRATE)* . RESTART_PACKAGES)* 2. BOOT_COMPLETED . (ACCESS_NETWORK_STATE | click | INTERNET | RESTART_PACKAGES | ACCESS_FINE_LOCATION)$^+$
Ggtracker. *Monitors received SMS messages and intercepts SMS messages* [2]	1. MAIN . READ_PHONE_STATE 2. SMS_RECEIVED . SEND_SMS
Ginmaster. *Sends received SMS messages to a remote server* [24]. *Downloads and installs applications without user concern* [24].	1. BOOT_COMPLETED . LOAD_CLASS 2. MAIN . SEND_SMS
Spitmo. *Filters SMS messages to steal banking confirmation codes* [4].	1. NEW_OUTGOING_CALL . READ_PHONE_STATE . INTERNET . (INTERNET | ϵ)
Zitmo. *Opens a backdoor that allows a remote attacker to steal information from SMS messages received on the compromised device* [4].	1. SMS_RECEIVED . SEND_SMS 2. MAIN . READ_PHONE_STATE 3. MAIN . SEND_SMS

Table 1. Learned unwanted behaviours versus human-authored descriptions.

A subjective comparison shows that these learned unwanted behaviours compare well to their human-authored descriptions. Also, they reveal trigger conditions of some behaviours, which were often lacking in human-authored descriptions. For example, the expression BOOT_COMPLETED.SEND_SMS denotes that after the device finishes booting, this application will send a message out; the expression UMS_CONNECTED.LOAD_CLASS means that when a USB mass storage is connected to the device, this application will load some code from a

library or a hidden payload; and the unwanted behaviour for Droiddream shows that if the phone state changes (PHONE_STATE), this application will collect information then access the Internet. Within the human-authored descriptions displayed in Table 1, only two behaviours are not captured by learned unwanted behaviours: "gain root access" for Droiddream and the behaviour of Spitmo.

5 Evaluation: Detecting New Malware

We are concerned with whether unwanted behaviours can help improve the robustness of malware classification. As we will show in Table 2, a linear classifier using API calls and permissions as input features, which are the most popular and the best performing input features for Android malware detectors [5, 8, 10, 14, 22, 36], performs badly on new malware instances (the testing set), although it has a very good classification performance on the validation set. In this section, we will show that unwanted behaviours improve the classification performance of new malware detection.

The training, validation, and testing sets are the same as those described in Section 1. Permissions and lists of API calls appearing in the code are extracted from these applications as input features to train classifiers as baselines. We construct behaviour automata for all applications, then apply methods discussed in Section 4.1 to learn unwanted behaviours from malware instances in the training set. We check whether the intersection between the behaviour automaton of the application in question and an (extended) unwanted behaviour is not empty. We collect these verification results as input features to train the target classifiers. For both baselines and target classifiers, we use L1-Regularized Logistic Regression [32] as the training method. The classification performance is reported in Table 2. The precision and recall are calculated as follows:

$$\text{precision} = \frac{tp}{tp + fp} \qquad \text{and} \qquad \text{recall} = \frac{tp}{tp + fn}\,,$$

where tp, fp, and fn respectively denote the true positives, false positives, and false negatives. This table confirms that:

- The unwanted behaviours dramatically improve the classification performance on new malware instances. The classification performance using API calls and permissions as input features is very good on the validation set, i.e., the precision and recall are respectively 93% and 98%. However, this is just over-fitting to the training set, since its performance on the testing set is bad, in particular, the precision is 65% and recall is 15%. This means that a lot of new behaviours cannot be captured by API calls and permissions. By using the verification results against unwanted behaviours as input features, we improve the precision to 73% and the recall to 66%, as shown in the row of "ext. unwanted for families".
- The generalisation from the unwanted behaviours to the extended unwanted behaviours helps improve the classification performance as well. We increase

the precision from 53% (in the row of "unwanted") to 69% (in the row of "ext. unwanted"). Although we lose several percent of recall, we get a better F_1-measure between precision and recall.
– Refining unwanted behaviours using the family names helps improve the classification performance of detecting new malware. The precision is increased from 69% (in the row of "ext. unwanted") to 73% (in the row of "ext. unwanted for families"), while maintaining the same recall. This refinement also helps reduce the number of features which are actually used in a linear classifier, i.e., totally 131 features were used, rather than 581 features.
– Combining syntax-based and semantics-based features results in over-fitting to the training dataset. By doing this, although the trained classifier can achieve the best classification performance on the validation dataset, its classification performance on the testing dataset is poor, in particular, the recall is as low as 7.5% (in the row of "all").

feature	*validation (2011–13)*		*testing (2014)*		#salient/#feature
training (2011–13)	precision	recall	precision	recall	
signature-based features (baselines)					
permissions	89%	99%	53%	21%	59/175
apis	91%	98%	61%	15%	1443/52432
apis & permissions	93%	98%	**65**%	**15**%	735/52607
semantics-based features (targets)					
unwanted	66%	91%	53%	74%	634/886
ext. unwanted	75%	87%	69%	66%	581/886
ext. unwanted for families	72%	72%	**73**%	**66**%	**131/131**
mixed features					
all	**95**%	**99.5**%	65%	7.5%	870/61149

Table 2. The classification performance using different features.

6 Conclusion and Further Work

To learn compact and verifiable unwanted behaviours from Android malware instances is challenging and has not yet been considered. Compared with manually-composed properties, unwanted behaviours, which are automatically constructed from malware instances, will be much easier to be updated on the changes of behaviours exhibited in new malware instances. To the best of our knowledge, our approach is the first to automatically construct temporal properties from Android malware instances. We show that unwanted behaviours help improve the robustness of malware classifiers, in particular, they dramatically increase the precision and recall of detecting new malware. These unwanted behaviours can not only be used to eliminate potentially new instances of known malware families, but also help people's understanding of unwanted behaviours exhibited in these families.

Some unwanted behaviours cannot be captured by our formalisation, e.g., gaining root access, in which specific commands are executed, and some are not

captured precisely enough, e.g., botnet controls, in which the communication between the app and the remote server has to be modelled. In further work, we want to extend the current formalisation to capture more sophisticated behaviours precisely. We will also try to combine the output of dynamic analysis, e.g., traces produced by CopperDroid [27] or MonitorMe [23], with that of static analysis to approximate applications' behaviours. It would be interesting to explore whether properties expressed in LTL are needed in the practice of Android malware detection and whether it is possible to learn them from malware.

The verification method adopted in this paper is straightforward and simple. More efficient and complex methods, e.g., the method discussed in [29] and model checking pushdown systems [18], will be considered in future.

The applications in current datasets were released between 2011 and 2014. More interesting comparison and study will be done when we get applications released in 2015 as another testing set.

Except for the unwanted behaviours, it is worth investigating whether other machine learning methods can help improve the robustness of malware classifiers, e.g., semi-supervised learning [15], the ensemble learning [13], the adaptive boosting [20], etc. We will also compare the robustness of popular machine methods, e.g., decision trees [26], SVM [31], naive Bayes, KNN [6], etc.

It is also interesting to study whether unwanted behaviours can convince people of the automatic malware detection.

References

1. Malware Genome Project. http://www.malgenomeproject.org/, 2012.
2. Forensic Blog. http://forensics.spreitzenbarth.de/android-malware/, 2014.
3. Juniper Networks. https://www.juniper.net/security/auto/includes/mobile_signature_descriptions.html, 2015.
4. Symantec security response. http://www.symantec.com/security_response/, 2015.
5. Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, 2013.
6. N. S. Altman. An introduction to kernel and Nearest-Neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.
7. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.
8. D. Arp et al. Drebin: Efficient and explainable detection of Android malware in your pocket. In *NDSS*, pages 23–26, 2014.
9. K. W. Y. Au et al. PScout: Analyzing the Android permission specification. In *CCS*, pages 217–228, 2012.
10. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *CCS*, pages 73–84, 2010.
11. P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification*, pages 168–182. 2010.
12. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.
13. L. Breiman. Random forests. *Mach. Learn.*, 45, 2001.

14. S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *WiSec*, pages 13–24, 2013.
15. O. Chapelle, B. Schlkopf, and A. Zien. *Semi-Supervised Learning*. The MIT Press, 2010.
16. K. Z. Chen et al. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.
17. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.
18. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, LNCS 1855, pages 232–247. Springer, 2000.
19. M. Fredrikson et al. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 45–60, 2010.
20. Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1), 1997.
21. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.
22. A. Gorla et al. Checking app behavior against app descriptions. In *ICSE*, 2014.
23. J.-C. Kuester and A. Bauer. Monitoring real android malware. In *Runtime Verification 2015*, 2015.
24. McAfee Threat Center. http://www.mcafee.com/uk/threat-center.aspx, 2015.
25. P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.
26. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.
27. A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In *European Workshop on System Security (EUROSEC)*, 2013.
28. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.
29. F. Song and T. Touili. LTL model-checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–431. 2013.
30. M. Spreitzenbarth et al. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.
31. I. Steinwart and A. Christmann. *Support Vector Machines*. Springer, 2008.
32. R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.
33. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.
34. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, July 2002.
35. C. Yang et al. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS*, 2014.
36. S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new Android malware detection approach using bayesian classification. In *AINA*, pages 121–128, 2013.
37. Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.